Bolt Beranek and Newman Inc.

Report No. 3972

(14) BBN-3972

(6)

Development of a Voice Funnel System:

(9) Quarterly Technical Report No. 1,
15 Jul[redacted]31 Oct[redacted]978

(10) R. /Rettberg

(15) MDA903-78-C-0356
ARPA Order-3653

30 Nov[redacted]978

(11)

(12) 32p.

Prepared for:
Defense Advanced Research Projects Agency

JOB

060 100 78 11 30 058

Report No. 3972 Bolt Beranek and Newman Inc.

DEVELOPMENT OF A VOICE FUNNEL SYSTEM

QUARTERLY TECHNICAL REPORT NO. 1
15 July 1978 to 31 October 1978

30 November 1978

This research was sponsored by the
Defense Advanced Research Projects
Agency under ARPA Order No.: 3653
Contract No.: MDA903-78-C-0356
Monitored by DARPA/IPTO
Effective date of contract: 1 September 1978
Contract Expiration date: 30 November 1980
Principal investigator: R. D. Rettberg

Prepared for:

Dr. Robert E. Kahn, Deputy Director
Defense Advanced Research Projects Agency
Information Processing Techniques Office
1400 Wilson Boulevard
Arlington, VA 22209

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) DEVELOPMENT OF A VOICE FUNNEL SYSTEM, QUARTERLY TECHNICAL REPORT NO. 1		5. TYPE OF REPORT & PERIOD COVERED Quarterly Technical 15 July 1978 to 31 Oct. 197
		6. PERFORMING ORG. REPORT NUMBER 3972
7. AUTHOR(s) R. Rettberg		8. CONTRACT OR GRANT NUMBER(s) MDA903-78-C-0356
9. PERFORMING ORGANIZATION NAME AND ADDRESS Bolt Beranek and Newman Inc. 50 Moulton Street, Cambridge, MA 02138		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ARPA Order No. 3653
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd., Arlington, VA 22209		12. REPORT DATE 30 November 1978
		13. NUMBER OF PAGES 29
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

UNLIMITED

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Voice Funnel, Digitized Speech, Packet Switching, Butterfly switch, Multiprocessor

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This quarterly Technical report covers work performed during the period noted on the development of a high-speed interface, called a voice funnel, between digitized speech streams and a packet-switching communications network.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

TABLE OF CONTENTS

1. Introduction 1

2. Processor Selection 3

2.1 Processor Candidates 3

2.2 Address Space 5

2.3 Z8000 Performance Evaluation 7

2.3.1 PDP-11 Benchmark 8

2.3.2 Z8000 Benchmark 10

2.3.3 A Discussion of the Benchmarks 11

2.3.3.1 Code Size 11

2.3.3.2 Calling a Global Routine 12

2.3.3.3 Returning from a Routine 12

2.3.3.4 Saving and Restoring Registers 12

2.3.3.5 Local Space Allocation and Deallocation . . 13

2.3.3.6 Setting Up Parameters for the Call 13

2.3.3.7 Making Parameters Accessible 13

2.3.3.8 A Typical Loop Iteration 14

2.3.4 Conclusions from the Benchmark 14

3. Processor Node Issues 17

3.2 Memory 18

3.3 Memory Management 19

3.4 Local I/O 22

1. Introduction

The combination of digital speech techniques with packet-switching technology can significantly increase the future voice transmission capability of the Department of Defense. To achieve this combination, however, requires a high speed interface between the digitized voice and the communications network. Bolt Beranek and Newman Inc. has been awarded a contract to design and develop a system which meets this need. The system, called a Voice Funnel, is a concentrator for digitized speech streams; it can combine on the order of 100 to 1000 streams with a high combined data rate (in the range of 2 to 20 Mbps).

The Voice Funnel requires high data rate support; many connections; and the ability to perform processing in order to control the machine and the data streams as well as to support sophisticated hardware and protocol interfaces to both the vocoders and the high bandwidth network. A new machine configuration which meets these requirements is described in BBN Report No. 3501, "A New Multiprocessor Architecture." We will assume familiarity with that report for the discussions here.

This Quarterly Technical Report covers the activity during the first few months of this project. During this time, most of the effort has concentrated on the design of the machine which is the foundation of the Voice Funnel. This machine consists of a number of independent processing nodes which are interconnected by a switch. The design of this switch is perhaps the most unique portion of this machine; it is certainly the most thoroughly studied and reported portion (see BBN Report No. 3501). During this quarter, we have made significant progress in defining the processing node and have continued our design of the switch. In light of the previous report, this Quarterly Technical Report concentrates on the processor choice and processing node design.

2. Processor Selection

The processor we have selected for use in the Voice Funnel is Zilog's Z8000. This machine has been announced with delivery expected before the summer of 1979. We have selected it because 1) it is a 16-bit microprocessor, 2) it has a reasonably high instruction execution rate, 3) it can manipulate a 23-bit virtual address space, and 4) it has a compatible memory management device.

A review of the competing processor candidates will be presented in this section, followed by a discussion of address spaces. Finally, since we find the Z8000 attractive as a processor, we will examine its instruction set and performance through one benchmark programming example.

2.1 Processor Candidates

There are three categories of candidates for the processor in the Voice Funnel:

1. Commercial minicomputer (e.g. PDP-11)
2. Commercial microprocessor
3. Custom microprogrammed processor

The advantage of a commercial minicomputer is the commitment on the part of the manufacturer: it is likely that compatible processors which are more powerful and/or lower in cost will be produced in the future. Furthermore, a complete line of compatible equipment and software is probably available. For microprocessors this has been true only over a very limited range. For example, the 8080 is now available at lower cost and in much faster versions, but the new architecture, the 8086, is only awkwardly compatible with the 8080. Similarly, Zilog with their Z8000 and Motorola with their M68000 have developed new architectures for their new products. A custom processor is even worse in this respect, since every product enhancement must be supplied by local effort.

The advantages of a commercial microprocessor are its small size, low cost, and multiple sourcing. The first two are particularly important since we expect to have many processors in our system, which puts a premium on a processor which has a high performance-to-size ratio as well as a high performance-to-cost ratio. We would also like to have a machine which adapts well to technological advancement. This is, after all, the age of LSI and even VLSI. It is clear that these technologies offer so much that it is hard to avoid taking advantage of them. The pattern of multiple sourcing is simply an added benefit. Multiple sourcing is not common in the arena of commercial minicomputers; the vulnerability that results from a single supplier can be serious. A custom machine can be of moderate cost (between a commercial mini and a micro) but its size is much larger than a micro.

The advantage of a custom microprogrammed processor comes from its being custom; it can be designed to fit well into the overall structure of the machine. If other available machines have serious system deficiencies, a custom processor design could be a better choice than trying to correct the problem or trying to work around it. Fortunately, as we will see later, there are available machines which are acceptable.

The second advantage of a custom microprogrammed processor comes from microprogramming. This permits efficient I/O device design and flexibility in the macro-level machine because of the very high rate of microinstruction execution. In raw cycle time, however, the microprocessor should eventually win since it eliminates inter-chip signals.

2.2 Address Space

Perhaps the most important criterion that we have used to select a processor is the need for a large address space. There are two address spaces involved: the physical address space and the virtual address space. The physical address space must be large enough to hold all of the memory in the machine. The virtual address space must be large enough to hold the entire process: its code and all of its data.

Experience has repeatedly shown that a 16-bit virtual address space is simply too small. Mapping hardware has often been suggested as a mechanism which can be used to expand the virtual address space. While a mapping mechanism is an appropriate mechanism for providing a virtual machine to user processes as in an operating system, its use for expansion of the virtual address space is incorrect. The reason for this is that it is too expensive and too awkward to set the maps prior to every memory reference. As a result, the program must be organized around the use of the maps. This is too large a burden to place on the programmer. As a result, programs end up being squeezed into the true virtual address space (as is the case with the UNIX kernel) or being chopped into many separate processes, with a serious increase in complexity and a serious loss of efficiency.

It is not clear how large these two spaces must be. An argument is made later that a 24-bit physical address space is adequate for the Voice Funnel, but it is very hard to know how large a virtual space is required; at least we know that an address space of 16 bits is too small.

Fortunately, Zilog's Z8000 provides both a large physical and a large virtual address space. Although this machine is a 16-bit processor, it has facilities which permit manipulation of 23 bits of virtual address. Furthermore, Zilog has announced a memory-management device which supports a 24-bit physical address space.

2.3 Z8000 Performance Evaluation

It has been pointed out that one of the major advantages of the Z8000 is its speed. To investigate this and to gain some insight as to the quality of the Z8000 we will investigate one benchmark program. From that benchmark we will get estimates of Z8000 speed and code size. As a point of reference, we will use the PDP-11.

The benchmark consists of a routine that allocates a large local array and calls a routine which returns the sum of that array. This benchmark exhibits calling costs, allocation costs, looping ease, and a reasonable amount of stack manipulation. The following BLISS-11 program is the benchmark:

```
MODULE BenchMark(OPTIMIZE,SAFE,NOUNAMES,FINAL,MAIN)
    = BEGIN

GLOBAL ROUTINE WordArraySum(Array,Size)=
   BEGIN
      WORD LOCAL sum,pointer;
      sum _ 0;
      pointer _ .array;
      DECR counter FROM .Size TO 1
         DO BEGIN
            sum _ .sum + ..pointer;
            pointer _ .pointer + 2;
         END;
      .sum
   END;

GLOBAL ROUTINE Bench=
   BEGIN
      WORD LOCAL vector[300];
      WordArraySum(vector,300);
   END;

END ELUDOM
```

From this benchmark we can hope to get reasonable measures of the costs of the following activities (the abbreviations in parentheses will be used in later tables):

Calling a global routine	(Call)
Returning from a routine	(Ret)
Register allocation	(Sav+Res)
Parameter setup	(Parm Set)
Parameter access	(Parm Acc)
A typical loop iteration	(Loop)
Local space allocation	(Loc Allc)
Local space deallocation	(Loc Dallc)

2.3.1 PDP-11 Benchmark

The BLISS-11 compiler produced the following code for this benchmark. The code is verbatim except for comments which have been added to the PDP-11 code for clarity.

```
; BLIS11 V.78095        Thursday 21-Sep-78 11:01.26     BENCH.B11
.PDP10
; 0001  MODULE BenchMark(OPTIMIZE,SAFE,NOUNAMES,FINAL,MAIN)
              = BEGIN
; 0002
.SBTTL  GLOBAL ROUTINE WordArraySum(Array,Size)=
; 0003  GLOBAL ROUTINE WordArraySum(Array,Size)=
; 0004     BEGIN
; 0005        WORD LOCAL sum,pointer;
; 0006        sum _ 0;
; 0007        pointer _ .array;
; 0008        DECR counter FROM .Size TO 1
; 0009           DO BEGIN
; 0010              sum _ .sum + ..pointer;
; 0011              pointer _ .pointer + 2;
; 0012           END;
; 0013        .sum
; 0014     END;
; 0015
.TITLE BENCHMARK

.CSECT BENC.C

WORDARRAYSUM:
        JSR     R$1,$SAV2       ; Allocate working space
        CLR     R$0             ; Clear sum
        MOV     12(SP),R$1      ; Get Parms: R1 <- Array pointer
        MOV     10(SP),R$2      ;            R2 <- Array size
        BR      L$4             ; Enter loop, check size zero
L$3:    ADD     (R$1)+,R$0      ; Loop: Add word, advance pntr.
        DEC     R$2             ;       Decrement counter
L$4:    BGT     L$3             ;       Repeat if more entries
        RTS     PC              ; Go home, (Get Regs: coroutine)

; ROUTINE SIZE:  12
```

```
.GLOBL  WORDARRAYSUM
.SBTTL  GLOBAL ROUTINE Bench=
; 0016  GLOBAL ROUTINE Bench=
; 0017     BEGIN
; 0018        WORD LOCAL vector[300];
; 0019        WordArraySum(vector,300)
; 0020     END;
; 0021

.CSECT BENC.C

BENCH:
        SUB     #1130,SP            ;Allocate: Local array, 300 words
        MOV     #2,-(SP)            ; Set Parms: Array pointer
        ADD     SP,@SP              ;   To array above this on stack
        MOV     #454,-(SP)          ;   Array size is 300 decimal
        JSR     PC,WORDARRAYSUM     ; Global routine call
        ADD     #1134,SP            ; Deallc: parms and local array
        RTS     PC                  ; Go home

; ROUTINE SIZE:  12

.GLOBL  BENCH
; 0022  END ELUDOM


.GLOBL $SAV2
.CSECT BENC.G

$BREG: .BLKW 1

; Size:  26+0
; Run Time:  0 Seconds
; Core Used:  12K
; Compilation Complete

.END BENCHMARK
```

2.3.2 Z8000 Benchmark

For comparison with the PDP-11 code, here is a hand-coded version of the benchmark for the Z8000. The version shown here is the most efficient way we could find to do the task, compared to, say, the cleanest way to do the task.

```
    WORDARRAYSUM:
            PSHL    RR2             ; Allocate: Reg for array pointer
            PSH     R4              ;           Reg for array counter
            CLR     R0              ; Clear the sum we are making
            LDL     RR2,16(SP)      ; Get Parms: Array pointer
            LD      R4,12(SP)       ;            Array size
            JR      CLR,L$2         ; If size is zero: skip loop
    L$1:    ADD     R0,(RR2)        ; LOOP: Add next array entry
            INCR    R2,#2           ;       advance the pointer
            DJNZ    R4,L$1          ;       decr cntr and loop?
    L$2:    POP     R4              ; Deallocate: Both
            POPL    RR2             ;             Regs
            RET     ALWYS           ; Go home, condition: always

    BENCH:
            SUB     SP,#(1130)      ; Allocate: Local word array[300]
            PSHL    (SP),SP         ; Set up Parms: Push and
            INCR    (SP),#4         ;               align array pntr
            PSH     (SP),#454       ;               and array size 300
            CALL    WORDARRAYSUM    ; Call global routine
            ADD     SP,#(1130+6)    ; Deall: Parms and Array
            RET     ALWYS           ; Go Home, condition: Always
```

2.3.3 A Discussion of the Benchmarks

We will discuss those two routines in the context of the categories mentioned in Section 2.3.

2.3.3.1 Code Size

Both routines are about the same size. The lack of a uniform dual-operand addressing structure for the Z8000 might be expected to cause larger code sizes in some situations. For example, both routines need to do some manipulations of parameters that are on the stack. The PDP-11 version manages this cleanly since memory can be treated identically to registers. The Z8000 manages it by having a convenient instruction, an increment instruction which is capable of an indirect reference. Since we have a pointer to the parameter, and the increment is small, we can do the address manipulation on the stack.

2.3.3.2 Calling a Global Routine

Both versions are effectively equivalent.

2.3.3.3 Returning From a Routine

Both versions are fine; the Z8000 has a conditional return that checks the state of the condition codes, an interesting idea.

2.3.3.4 Saving and Restoring Registers

The PDP-11 version uses a standard PDP-11 technique for saving registers: a global routine is called that does the save and then does a coroutine return so that when the routine finishes the registers will be restored automatically. In all cases except the save of one register, on all PDP-11 models, this technique is faster and produces less code. A more traditional technique would look like:

```
MOV  R1,(SP)-   ; Allocate: Register
MOV  R2,(SP)-   ;          and another
<routine body>
MOV  +(SP),R2   ; Deallocate: Register
MOV  +(SP),R1   ;            and another
```

We will assume this is the technique used by the PDP-11 version of the benchmarks. It is interesting, however, to note that coroutine linkages are not a natural programming primitive provided by the Z8000.

The Z8000 register saves are now identical to the PDP-11 technique. The existence of a block register transfer instruction is somewhat irrelevant since it copies upward in memory while the stack grows downward. This means to use it you must first align the stack pointer. In most cases it is easier to use push and pop instructions.

2.3.3.5 Local Space Allocation and Deallocation

In both versions this is a simple addition or subtraction from the stack pointer.

2.3.3.6 Setting Up Parameters for the Call

The PDP-11 version proceeds by pushing the offset from the stack pointer to the array onto the stack and then adding the stack pointer to it.

The nice quality of this operation is the addition that is done on the stack. This single addition might have cost a register allocation, a move of the stack pointer, an addition, and then the deallocation of the register. The Z8000 version manages the same feat via the increment instruction.

2.3.3.7 Making Parameters Accessible

In these examples this category means getting the parameters off the stack and into the registers. Both versions have little trouble with this.

2.3.3.8 A Typical Loop Iteration

The loop consists of four parts: addition, pointer advancement, counter decrement, and conditional branch. The PDP-11 version is able to combine the first two, while the Z8000 version is able to combine the second two.

The Z8000 version is more attractive in some ways. The PDP-11 attitude was that the branch would be able to use the last data manipulation result via the condition codes to drive the branching decision. This has not turned out to be true and has often caused practices such as placing a zero value on the end of arrays to act as a flag. The decrement and skip on zero instruction provided by the Z8000 is better; it would be even better, of course, if it had some range of increments, and condition codes.

2.3.4 Conclusions from the Benchmark

The tables below compare three machines: the LSI-11, the Z8000 running in nonsegmented mode, and the Z8000 running in segmented mode. The LSI-11 is a machine that addresses 2^16 bytes naturally, as is the Z8000 running in nonsegmented mode; the Z8000 running in segmented mode can address 2^23 bytes and hence is slowed somewhat by having to manipulate larger addresses.

The first table below shows code size, memory fetches, and execution times for all three machines. All fetch counts are in

16-bit words, all code sizes are in 16-bit words, and all execution times are in microseconds.

	LSI-11			Nonsegmented Z8000			Segmented Z800		
Category	Time	Size	Fetch	Time	Size	Fetch	Time	Size	Fetch
Call	6.95	2	2	1.75	2	2	2.5	3	3
Ret	5.25	1	2	1.5	1	2	2.25	1	3
Sav+Res	20.3	4	8	7.0	4	8	8.5	4	10
Parm Set	19.6	5	9	6.0	4	8	6.75	4	9
Parm Acc	12.3	4	6	5.0	4	6	6.5	4	7
Loop	12.6	3	4	3.75	3	4	3.75	3	4
Loc Allc	4.9	2	2	1.75	2	2	1.75	2	2
Loc Dall	4.9	2	2	1.75	2	2	1.75	2	2

The next table contains the same values converted to percentages of their LSI-11 equivalents.

	Nonsegmented Z8000			Segmented Z8000		
Category	Time	Size	Fetches	Time	Size	Fetches
Call	25	100	100	36	150	150
Ret	28	100	100	43	100	150
Sav+Res	34	100	100	42	100	125
Parm Set	31	80	89	34	80	100
Parm Acc	38	100	100	49	100	117
Loop	29	100	100	29	100	100
Loc Allc	36	100	100	36	100	100
Loc Dallc	36	100	100	36	100	100
Averages	32%	98%	98%	38%	104%	118%

In summary:

- The code density of the Z8000 is very similar to that of a PDP-11.

- The execution speed of the Z8000 is about 3 times that of an LSI-11.

- It is straightforward to code those things that programs do most often.

- A few nonuniformities in the Z8000's instruction set will make code generation more difficult than for the PDP-11, but in general not too much more difficult.

3. Processor Node Issues

The processing node consists of 5 pieces: a processor, memory, memory management, local I/O, and the switch interface. We will consider each of these components in turn.

3.1 The Processor

Although it now seems very likely that the initial design will use a Zilog Z8000 processor, it is useful to examine not just the characteristics of that specific processor, but also those attributes of an abstract processor that might affect the design of the processing node, for example:

- How fast is it?
- Does it multiplex its address and data or are they presented simultaneously?
- Does the processor do its own refreshing or will extra hardware be required?
- How wide is the word?
- How wide is an address?
- Are there multiple protection modes?
- How does the processor interact with I/O?
- Are I/O addresses in a separate address space?
- What are the provisions for independent DMAs and how do these interact with the processor and memory manager?
- How does the interrupt system work?

There are some primitives which are specifically important for a multiprocessor such as a facility for "locking", e.g., a test and clear instruction or some semaphore primitives. At least some kind of read-modify-write bus operation is needed in order to guarantee indivisibility.

An important issue has to do with how closely tied the processor is to its memory. In this system, the processor may have to wait significant periods of time for the datum fetched. Most processors include some kind of "wait" input, but these do not always operate easily or smoothly. In this case, we have the further complication that the wait time may be quite significant (as requests traverse the switch). If the wait state interferes with real-time operations, e.g., refreshing memory, there may be a problem. Those processors which "wait" by stopping their clock may have a maximum wait time imposed by the constraints of internal dynamic memory.

3.2 Memory

The memory in each processor node serves not only as the local memory for the processor and I/O in that processor node but also as a portion of the global memory space which is accessible from any processor node.

Perhaps the largest problem in designing the memory for the processor node is permitting flexibility in memory size without mounting the memory components on a separate board.

The technology used for the memory circuitry is rather straightforward. Dynamic semiconductor memories are dense and inexpensive. The problem of refresh is solved by the Z8000 processor, leaving only timing and error control.

Error control is the most difficult question. The current industry wisdom holds that large dynamic memory systems need error detection AND correction (EDAC). A Voice Funnel, however, can have many independent processor nodes, each with its own memory modules. If one or two are down in the system as a whole, that is not a big loss. An appropriate memory management scheme should be able to isolate the application software from memory failures. This simplifies our requirement to only a simple method of detecting memory errors, e.g., parity. We do not have to spend the relatively larger cost increment for single bit correction (5 bits on a 16-bit word is a 32% increase in memory price alone, without considering the correction logic itself or any access time delay imposed).

3.3 Memory Management

The purpose of memory management is to separate the virtual address space seen by the processor from the actual memory configuration, the physical address space. The memory manager serves as an interface between these two spaces, translating (mapping) virtual addresses into physical addresses. The time of the mapping is also a convenient one to do some ancillary functions, like protection. There are many reasons for providing such a separation but the main idea is to isolate the programmer from the vagaries of the hardware. The program should be insensitive to what processing nodes or memory modules actually exist, or even where a particular piece of data actually resides. The programmer tags a "thing" (code page, data structure, word of memory, etc.) with an invariant name (its virtual address). The job of the memory management system is to make sure that when that virtual address is referenced, it is directed to the proper place.

The system that we choose is a simple one. The physical address (as output by the memory manager) is sufficient to locate any word in any processing node in the entire machine. That is, encoded within it is a processing node number and a location within the memory of that processing node. A process selects which objects it needs to reference and, with the assistance of the operating environment, the local memory manager is set up with the physical addresses of those objects. When an access is made, the memory manager converts the virtual address to a physical address and simple address recognition logic decides if that is located in this processing node. If not, the physical address is handed to the switch interface which tries to get that data by sending out an appropriate transaction on the switch. When this reaches the destination processing node, the incoming request for a word has a physical address contained in the requesting transaction. Thus the switch interface can directly access the local memory for the word and return it.

A complication arises should the operating environment decide to move an object to a different physical address. In that case, all the memory managers which contain a pointer to this object must be updated. The operating environment can do this by informing each processing node each time it wishes to make such a change. If we were planning a classic paging system, with pages of memory moving about all the time, this would not be a good scheme. As it is, we do not envision the need to move things around very much, and so this safe, easy to understand mechanism should not be a burden.

Zilog is planning to introduce a memory management chip to perform some of these functions. It accepts 23-bit virtual addresses: 7 bits of "segment" and 16 bits of displacement within that segment. The output is a 24-bit physical address. (All addresses are in terms of bytes.) For each segment, an internal memory contains:

16 bits of offset (in units of 2**8 bytes)

8 bits of size (in units of 2**8 bytes)

8 bits of attributes:

R/W vs. R only
system vs. user
execute vs. data
exclude DMA
invalid entry
segment changed
segment referenced
something else

Although the basic functionality of this device seems to meet our needs, there is some question about whether the virtual and physical address spaces are large enough. A 24-bit physical address seems like a lot, but if we have 64K words of memory in each node, that limits us to 128 nodes. This is certainly adequate for our immediate needs and looks as if it can easily be expanded if future needs dictate. On the virtual side, we are limited to 128 active segments at any time. Our current expectation is that this will be adequate. In any case, the address spaces offered by the Z8000 and its memory manager are much larger than anything offered by other microprocessors.

3.4 Local I/O

The key idea here is to lever our effort by taking maximum advantage of the devices offered by the processor's manufacturer. By using the units that the processor expects, we benefit from the manufacturer's efforts to ensure that they integrate well and work together properly.

In our case, Zilog is developing a line of I/O devices which can be attached to the Z-Bus (the Z8000 I/O bus). For the many vocoder interfaces, we will probably use the Z-SIO, a serial I/O device capable of interfacing asynchronous, bit-stuffing synchronous, and byte-stuffing synchronous protocols. It should be possible to program this device to support a very wide range of vocoder types. Should we need some very low level processing associated with an I/O interface, we can use the Z-UPC, an integrated processor, memory, and I/O chip (actually a Z-8) that is interfaced to the Z-Bus.

For the high bandwidth connection, the Z-SIO is too slow since its maximum rate is only 800 kbps. Probably the best choice is some custom logic attached to a Z-MBU, a FIFO with interface logic which also attaches to the Z-Bus.

3.5 Switch Interface

Before we discuss the switch interface, it is important to distinguish between two methods of using the switch: one is word-at-a-time, the other is via messages. In a tightly coupled multiprocessor, the processor often wants to access distant memory in the same way it accesses a word in local memory. Notice that in this model, the processor is passive as the transaction is manipulated.

In a message-based system, a processing node is a self-sufficient unit which actively communicates with other processing nodes. Processor instructions (or microcode) explicitly set up a message and send it out. A processing node would presumably always have a buffer set up for incoming messages. Upon receipt of one, the processor would be interrupted, and would explicitly deal with the message. As an example, these messages could be of the form "please give me word x". Presumably, it would be more efficient to send requests at a higher level of abstraction, i.e. ask to do more complicated things.

Let us look at what the word-at-a-time model implies. First consider a processor request. The memory manager must decide whether the word desired is in local memory (or I/O) space. If not in this processing node, it signals the switch interface that this is a request which it must deal with. The processor is told to wait and the switch interface forms up the message. The latter procedure consists of taking the physical address output by the memory management and concatenating it with some appropriate leader and trailer information. When the result returns (we know it is a result from a previous access because of some type codes), the switch interface hardware hands the result to the processor and allows the processor to continue.

A message model, on the other hand, implies some hardware which is more like a DMA device. The processor would tell the switch interface the limits of the message in memory. The switch interface would then send an appropriate leader and then pull the words out of memory as they were needed to be sent out to the switch. A receiving side of the interface would also have buffer pointers into memory; when a transaction started arriving from the switch, words would be dumped into memory. On completion, the processor would be interrupted.

We have discussed the message type switch interface as if it all lived in physical space. Another possibility is that the DMA-like device is just like any other I/O device and sends its "requests" (to send words out the switch interface) through the memory management box.

So what jobs must the switch interface do? (Let us assume for the moment that we are using the word-at-a-time model.) There are 3 kinds of transactions:

1. The processor accesses distant memory.
2. A result from an access of type 1 returns and is delivered to the processor.
3. A request from some other processing node requests a word from this local memory.

Types 1 and 2 interlock completely; only one request can ever be outstanding at any time and a type 1 can never be issued until the matching type 2 for the previous type 1 occurs. (N.B. This is NOT so true if there are master-type I/O devices and the processor can be held up without tying up the bus, e.g., it uses a transaction bus.) Type 3 accesses, however, can happen at any time and in any quantity. Note also that type 3 accesses conflict with both types 1 and 2. There must be some internal flow control mechanism which does not allow an access if it conflicts with one already in progress.

The memory also sees another kind of access which we can add to the above list:

4. Processor accesses local memory.

Notice the small number of possible accesses between the processor, the memory, and the switch interface. Each of the three destinations has at most two sources as follows:

Destination	Source	Type
Memory	Processor	4
Memory	Switch(R)	3
Switch(T)	Processor	1
Switch(T)	Memory	3
Processor	Memory	4
Processor	Switch(R)	2

Since there are only a small number of combinations and it is easy to state which actions are not allowed when others are occurring, the hardware which implements the switch interface can be very simple.

How does a message transaction fit in to this description? The switch interface must start the transaction through the switch, and then continue to get (or put) words from (into) memory as the message flows through. How does the speed of the switch compare with that of memory? There may well be delays at the beginning of sending a message, as the header encounters conflicts and tries to blaze a path through the switch, but once that has occurred, the message can then flow. At this point, if the switch is slower than memory, the memory must be idle. If the switch is faster than memory, we have accomplished nothing, except creating the necessity for a flow-control mechanism and causing more conflicts in the switch. Thus we can assume that the memory and switch speeds are closely matched.

A corollary to this is that when a message is "streaming", no other accesses can occur to the processing node's memory. We must make sure that this does not affect other real-time dependent functions, such as servicing a rotating memory, etc. We have a particular problem with memory refresh, but if the memory design allows, we can take advantage of the fact that switch I/O will be to consecutive addresses and thus may be able to perform the refresh directly.

At this time, it seems that both the word-at-a-time model and the message model are relevant. The word-at-a-time model permits tight coupling and low delay while messages are critical to high bandwidth transfers; it is our current intention to support both models in the hardware.

DISTRIBUTION OF THIS REPORT

Defense Advanced Research Projects Agency

Dr. Robert E. Kahn (2)

Defense Supply Service -- Washington

Jane D. Hensley (1)

Defense Documentation Center (12)

Bolt Beranek and Newman Inc.

Library

Library, Canoga Park Office

R. Brooks

P. Castleman

W. Clark

F. Heart

D. Hunt

B. Hyde

M. Kraley

R. Rettberg

E. Starr

D. Walden

E. Wolf

C. Wyman